Reasoning Support for Ontology Design

Carsten Lutz, Franz Baader, Einrico Franconi, Domenico Lembo, Ralf Möller, Riccardo Rosati, Ulrike Sattler, Boontawee Suntisrivaraporn, Sergio Tessaris

http://www.tonesproject.org/

Abstract. The design of comprehensive ontologies is a serious challenge. Therefore, it is necessary to support the ontology designer by providing him with design methodologies, ontology editors, and automated reasoning tools that explicate the consequences of his design decisions. Currently, reasoning tools are largely limited to the reasoning services (i) computing the subsumption hierarchy of the classes in an ontology and (ii) determining the consistency of these classes. In this paper, we survey the most important tasks that arise in ontology design and discuss how they can be supported by automated reasoning tools. In particular, we show that it is beneficial to go beyond the usual reasoning services (i) and (ii).

1 Introduction

The purpose of an ontology in computer science is to formally and unambiguously describe the relevant notions of a domain. This formalization of the domain's terminology then constitutes the basis for a shared and generally accepted understanding of the domain. Ontologies are used by humans to support communication among peers, as they make the notions used in such communication precise. They are used in information technology to facilitate content-based access and integration of information systems by assigning a precise meaning to the data stored in such systems. For successfully playing this important and critical role in communication and information provision, it is of utmost importance that an ontology is well-designed, clearly structured, and easily understandable to all relevant parties.

It is uncontroversial that the design of comprehensive ontologies in expressive ontology languages such as OWL is a challenging task [27]. The encountered problems include (but are not limited to) the following:

– Just like software design, ontology design involves a huge number of design decisions which range from fundamental ones such as "how to structure the modelled domain into subdomains?" to very concrete ones such as "should the color red be represented as a class, an individual, or a datatype"? Anticipating the consequences of such design decisions requires very firm knowledge of the application domain and ontology language.

– In expressive ontology languages such as OWL, there can be complex interactions between different class definitions or even between different parts of the same class definition. Due to the declarative style of class definitions in ontology languages, the ontology designer cannot rely on some execution model as in

software programming to guide his intuition about the effects of his definitions. Consequently, subtle interactions between class definitions can be difficult to notice.

These problems are aggravated by the fact that large ontologies are often designed by a group of interacting designers that have to establish common knowledge of the design decisions and intended structure of the ontology.

To master the complexities of ontology design, the ontology designer should be supported by ontology design methodologies, ontology editors, and automated reasoning tools. In this paper, we focus on the latter. Compared to software design, the use of automated reasoning tools plays a more important role in ontology design. The reason for this is two-fold. First, automated reasoning *should* be used: in software engineering, it is possible to run systematic tests to evaluate the correctness of a newly designed program. Since the possible ways in which an ontology will be used are often unknown at ontology design time, systematic test suits are usually much more difficult to attain in ontology design than in software design. Automated reasoning can be used to make the consequences of a certain ontology design explicit, and thus allows to evaluate the ontology's correctness without depending on concrete test cases. Second, automated reasoning *can* be used. A distinguishing feature of ontology languages such as OWL and description logic (DL) is that they have been carefully designed so that automated reasoning about ontologies is feasible in practice. In particular, ontology languages from this class are usually not Turing complete like programming languages, and many of the relevant reasoning tasks are decidable.

Traditionally, reasoning support for ontology design is largely limited to checking the consistency of classes (i.e., determining whether a class can have any instances) and to computing the subsumption hierarchy (a hierarchy which arranges the classes according to the subclass relationship). While these reasoning services are very helpful in particular for making implicit consequences of an ontology design explicit, we believe that reasoning support can go much further. The purpose of the current paper is to survey important tasks that frequently occur during ontology design and that can be supported by automated reasoning tools. For each task, we provide a general description, discuss how automated reasoning tools can support the designer, and provide references to the literature. The material presented in this paper has been developed within TONES ("Thinking ONtologiES"), an Information Societies Technology 3-year STREP FET project financed within the European Union 6th Framework Programme. The TONES project aims at enhancing and developing novel reasoning services for ontology design, maintenance, usage, and interoperation.

2 Authoring Class Definitions

One of the most central activities during ontology design is the formulation of the class definitions that constitute the ontology. Intuitively, authoring class definitions in ontology design is comparable to writing program code in software development.

Task Description Experience in ontology design shows that even users who are experts in both the ontology language and the domain to be formalized often find it challenging to formulate adequate and correct class definitions. The main reason for this difficulty is that class definitions are specified in a declarative way using an ontology language with a rich semantics such as OWL. Due to the declarative style of class definitions, the ontology designer cannot use some execution model to guide his intuition about the effects of design decisions. Due to the rich semantics, it can be far from obvious to the author of a class definition which implicit consequences are resulting from the explicit definition that he has produced.

The most dramatic such consequence is that the designed ontology is inconsistent, which means that there is no model (i.e., possible state of the world) that matches the class definitions contained in the ontology. Often, an inconsistent ontology is the result of adding a new class definition that interacts with the existing ones in an unintended way. An inconsistent ontology always indicates a serious modelling flaw and should be automatically detected and then reported to the ontology designer for resolution.

Another common and undesired consequence of an ontology is that a single class becomes inconsistent, i.e., this class cannot have any instances. Note that all classes contained in an inconsistent ontology are inconsistent w.r.t. this ontology, but there may be inconsistent classes in an otherwise consistent ontology. In this sense, inconsistent classes are a less severe problem than an inconsistent ontology. Nevertheless, they usually indicate a modelling mistake that requires inspection by the ontology designer.

A third kind of implicit consequence that can arise is that some class turns out to be a subclass of another class. Such implicit subclass relationships may or may not be intended. In any case, implicit such relationships should be detected and reported to the designer for inspection.

It is important to observe that the problem of implicit consequences becomes even more serious due to the fact that the class definitions contained in an ontology can interact in a serious, yet subtle way. For example, when authoring a definition for the class *Heart*, the author will refer to many other classes such as *Vessel*, *Tissue*, and *Blood*. The class definitions of the referred concepts usually interact with one another and with the class definition currently devised. Understanding such interactions often turns out to be the most time-consuming task when authoring class definitions.

Reasoning Support The automated detection of implicit consequences of an ontology is a key research issue in the field of description logic [2]. DL is one of the main roots of ontology languages such as OWL and in fact, one of the major design criteria of OWL was to control the expressive power of the language such that automated reasoning about the implicit consequences of an ontology is possible in practice [11]. Therefore, it is hardly surprising that reasoning tools for computing the consequences of ontologies formulated in OWL and other DLs are readily available. Well-known examples include RACER [10], FaCT++ [25],

and Pellet [23]. Such reasoners usually concentrate on detecting the three kinds of consequence discussed above: inconsistent ontologies, inconsistent classes, and subclass relationships.

The detection of implicit subclass relationships is closely related to *classification*, which is one of the most important reasoning services for ontology design. The purpose of classification is to compute the *subsumption hierarchy*, i.e., to arrange all the classes of the ontology in a hierarchy w.r.t. the subclass relationship. It is not hard to see that classification can be implemented by carrying out multiple subclass checks. The computed subsumption hierarchy can be displayed to the user, e.g. by an ontology editor such as Protégé [8]. It provides the user with a visualization of the ontology's structure and is the premier way to navigate and access the ontology.

The literature on the computational properties and implementation of reasoning services based on consistency and the subclass relationship is vast and a comprehensive overview is out of the scope of this paper. Therefore, we confine ourselves with a reference to the Description Logic handbook and the references therein [2].

3 Error Management

The purpose of the reasoning services described in Section 2 is to inform the ontology designer about ramifications of his modelling. Obviously, this is most useful when the reasoner finds ramifications that were not intended by the designer. However, understanding *why* such unintended ramifications hold can be a rather difficult task itself, in particular if the ontology is of large size and intricate structure.

Task Description This observation suggests that automated reasoning support should go one step further: additionally to reporting ramifications, it should assist the ontology designer in understanding and resolving them. Basically, support of the latter kind can take three different forms:

Pinpointing. To pinpoint an unintended ramification means to identify those parts of an ontology that are the source of this ramification. For example, a problem in an ontology that is caused by only two or three interacting class definitions may result in hundreds or even thousands of classes becoming inconsistent [22]. In such a case, it can be very difficult to identify the class definitions that actually cause the problem. Clearly, automatically pinpointing those definitions is a tremendous aid to the ontology designer for removing the problem. Still, pinpointing is only the weakest form of support.

Explanation. Explaining an unintended ramification means to provide a convincing argument that is understandable to the ontology designer and shows why the unintended ramification holds. Note that this is more than just identifying the concept descriptions that participate in the ramification since it also involves explaining the interplay between these descriptions and why they imply

the ramification at hand. Usually, explanation is much a more difficult task than pinpointing because for complex ontology languages such as OWL, automatically generating explanations that are of an acceptable length turns out to be diffcult.

Automatic Revision. Here, the idea is not only to explain the reasons for the unintended ramification to the designer, but also to make concrete suggestions for how to resolve the problem. This is a challenging problem because there are often a lot of options to resolve an inconsistency or undesired subclass relationship, but usually only very few of these options are intuitively acceptable. The suggestion for problem resolution can again be explained in an understandable way to the ontology designer.

Reasoning Support Pinpointing was first taken serious as a reasoning problem in DL-like languages in [22], which introduces "minimal unsatisfiability-preserving sub-ontologies (MUPS)". Intuitively, if a class C is inconsistent w.r.t. an ontology $\mathcal{O}$, then a MUPS for C and $\mathcal{O}$ is a minimal subset $\mathcal{O}'$ of the class definitions in $\mathcal{O}$ such that C is inconsistent w.r.t. $\mathcal{O}'$. Thus, a MUPS pinpoints the concept descriptions in $\mathcal{O}$ that are the source of the inconsistency of C. The related notion of MIPS identifies a minimal subset of an ontology in which *some* class is inconsistent. This can be useful if inconsistency of one class leads to inconsistency of many other classes, and it is unclear which class definition to correct when looking at MUPS. The approach of MUPS is further extended into the direction of automatic revision in [21], where it is shown how to convert MUPS into minimal sets of class descriptions that can be removed to make an inconsistent class in an ontology consistent. The approach of MUPS and automatic revision based on MUPS has been further developed and extended to more expressive logics in [12, 13]. It has been implemented in the SWOOP ontology editor.

Results on explanation and revision are much more sparse. Explanation has first been considered for relatively small fragments of OWL that admit so-called structural subsumption algorithms, which decide the subclass relationship between classes by normalizing and comparing the class definitions [17, 16]. Explaining the results returned by modern tableau-based reasoners for expressive DLs such as RACER and FaCT is a much more difficult task. To the best of our knowledge, the only available technique is the one presented in [5], where explanation is based on deductions in a sequent calculus that "follows" the computation performed by the tableau algorithm. Apart from the not very sophisticated technique based on MUPS, there seem to be hardly any approaches for addressing the automatic revision of ontologies. An exception is [18], where belief revision techniques are applied to description logic. However, that approach considers a scenario that is quite different from ontology design.

4 Stepwise Extension and Refinement

Ontologies show their full potential when being used to describe broad application domains such as medicine [20, 6, 24] and biology [26]. The terminology of such broad domains is usually structured into a number of subdomains, i.e., sets of notions that are closely interrelated, but not very closely related to most other notions in the domain. Of course, subdomains can again be structured into subdomains, etc. This structure of the terminology is reflected by the class definitions in the ontology. For example, the SNOMED medical ontology [24] includes subdomains for anatomy, diseases, and treatments. Another structure that can often be found in ontologies is the division into a *foundational part* that describes notions of a general nature and a *domain part* that describes notions specific for the modelled domain [19]. In the following discussion, we will treat the foundational part of an ontology in the same way as a subdomain.

Task Description A typical pattern in ontology design is to first concentrate on the class descriptions for one subdomain, then extend the ontology to an additional subdomain, and so on. Thus, the designer constantly extends the ontology with new subdomains and the overall ontology is built by *stepwise extension*. It is worth noting that this stepwise extension pattern can usually not be followed in a strict way. In particular, it is often not possible to describe the terminology of a subdomain without referring to other subdomains at all. Thus, it frequently happens that during the modelling of one subdomain, initial and usually very coarse class definitions for other subdomains are introduced. When these other subdomains are modelled in full detail, the initial class definitions need to be *refined*. Thus, stepwise extension and refinement are among the core tasks of ontology design.

When the ontology designer adds a new subdomain to the ontology or refines an existing one, he wants to be sure that the existing class descriptions of other subdomains are not compromised. This is particularly desirable because establishing the correctness of a (subdomain of an) ontology is a difficult and time-consuming process, and the designer should not be forced to repeat this process after each extension/refinement. This indicates the high benefit that can be expected from automated reasoning tools that are capable of detecting the consequences that the addition or refinement of a new subdomain has on other subdomains.

Reasoning Support For providing automated reasoning support, the intuitive notion of a subdomain "being compromised" by the addition of another subdomain has to be made precise. A weak interpretation is that a subdomain is compromised if there is a change in the subsumption hierarchy of the classes that are defined in this subdomain. This can clearly be detected using the reasoning tools mentioned in Section 2, which are capable of (re)computing the subsumption hierarchy. A stronger interpretation has been proposed in [1, 9], where the notion of a *conservative extension* is used.

The central idea of using conservative extensions for detecting harmful ontology extensions and refinements is to partition the vocabulary of the ontology according to the subdomains, i.e., each class, property, etc is associated with the subdomain to which it belongs. In the case of very general classes and properties such as "has-part", the corresponding subdomain may be the foundational part of the ontology. Let $\mathcal{O}$ be an ontology and let $\mathcal{O}'$ be obtained from $\mathcal{O}$ by adding the class descriptions for an additional subdomain. Moreover, let $\mathcal{V}$ be the vocabulary of a subdomain S in $\mathcal{O}$. Then the extension of $\mathcal{O}$ to $\mathcal{O}'$ does not compromise S if $\mathcal{O}'$ is a conservative extension of $\mathcal{O}$ w.r.t. the vocabulary $\mathcal{V}$, i.e., if there are no class descriptions C_1 and C_2 formulated using the vocabulary $\mathcal{V}$ such that C_1 is subsumed by C_2 w.r.t. the ontology $\mathcal{O}'$, but not w.r.t. $\mathcal{O}$. Observe that the class descriptions C_1 and C_2 need not occur in $\mathcal{O}$ and $\mathcal{O}'$, and thus this interpretation of being compromised is stronger than the weak interpretation proposed above.

A first investigation of reasoning procedures for conservative extensions in OWL-like languages has been carried out in [9, 14]. It turns out that the computational complexity of deciding conservativeness is much higher than that of the standard reasoning services from Section 2. In fact, conservativeness is still decidable for non-trivial fragments of OWL such as $\mathcal{SHIQ}$, but undecidable for full OWL-DL [14]. To the best of our knowledge, implemented reasoning tools for deciding conservativeness in OWL-like languages are not (yet) available. As an alternative to deciding conservativeness, it has been proposed in [7] to impose certain syntactic restrictions on well-designed ontologies. These restrictions would ensure that the extension or refinement of an ontology is *always* a conservative extension. In such a normative approach, a reasoner for deciding conservativeness is not required.

In mathematical logic and software there exists also a second notion of conservative extensions. While the conservative extensions introduced above can be described as proof-theoretic, the second variant is model-theoretic and defined in terms of model extensions [15]. However, it is shown in [14] that model-theoretic conservative extensions are undecidable even for quite small fragments of OWL-DL.

5 Generating Class Definitions

In Section 2, we have assumed that the ontology designer comes up with a definition of the class that he wants to add to the ontology, and then uses automated reasoning services to verify that the class definition does not interact in unexpected ways with the existing class definitions in the ontology. However, there are several situations in which it is desirable to generate a class definition automatically in order to support the ontology designer.

Task Description There are at least two reasons for why an ontology designer may find it difficult to produce an appropriate definition for a new class that is to be added to the ontology. The first reason is lack of knowledge about the

application domain. If this is the case, the ontology designer may have only a vague idea about the new class to be defined. In particular, it may not be clear which properties of the instances of the class are characteristic and should be included in the class definition. And second, the ontology designer may lack proficiency in the ontology language. In this case, despite having a clear idea of the class to be defined, he may not be able to come up with an adequate class definition.

In the described situations, an automated reasoning tool can assist the ontology designer by automatically generating an initial candidate class definition that can then be reviewed and manually refined be the ontology designer. There are mainly two sources of information about the new class that can be used as an input to the automated generation of the class description:

1. It is often the case that the ontology designer knows the place in the subsumption hierarchy where the additional class is supposed to appear. In this case, information about the new class can be deduced from intended position in the hierarchy.
2. The ontology designer usually knows a number of typical instances of the class to be defined. If this is the case, he can describe these instances in an appropriate language (e.g. using OWL individuals), and the resulting descriptions can be used to deduce information about the new class.

A particular advantage of the automatic generation of class definitions is that the generated definition will reflect the modelling decisions that have already been made in other parts of the ontology and that are relevant for the description of the new class. Hence, starting with an automatically generated class definition decreases the risk that the designer makes incoherent modelling decisions.

When deriving information about the new class using the intended position in the subsumption hierarchy, the intended subclasses are particularly valuable. Exploiting a known superclass usually boils down to simply stating the superclass relationship. In contrast, the definitions of the subclasses provide us with much more information because a description of the commonalities of all subclasses can be viewed as a candidate definition for the new class.

For this reason, the automatic generation of class definitions is particularly appropriate when the ontology is designed in a bottom-up fashion, i.e., by starting with the most specific classes of the application domain and working towards the more general classes. In this case, the subclasses of a new class have already been defined and can be used to generate the new class definition.

Reasoning Support The description logic reasoning problems *least common subsumer (LCS)* and *most specific concept (MSC)* can be used to autmatically generate class definitions in the described way. More precisely, the LCS operation is used to compute the definition of a class that is a superclass of a set of given subclasses, and that is the least general class with this property. Intuitively, such a class represents the commonalities among the classes in the set to which the LCS operation was applied. The MSC operation can be used to convert the

description of a single instance into a class definition. The LCS operation can then extract the commonalities of the resulting class definitions into a single class definition.

The literature on LCS and MSC is too large to be reviewed here in detail. Therefore, we refer only to the recent handbook article [3]. Currently, the main limitation regarding LCS and MSC is that they are usually considered in the context of relatively weak ontology languages. For example, the LCS is only meaningful in ontology language that do not provide for disjunction, and the MSC need not exist in most ontology languages. Thus, the prime future research issue is to extend LCS and MSC to more powerful languages and ontology formalisms. The work in [4] gives first ideas about how to employ the LCS in the context of ontologies that are formulated in ontology languages which include disjunction.

6 Discussion

We have identified important tasks that play a fundamental role in ontology design and can be supported by automated reasoning tools. Instead of trying to establish an exhaustive list of tasks, we have focussed on a number of selected tasks that we consider most relevant and general. Other important tasks arise when an ontology is designed with a specific application in mind. For example, if the ontology is constructed with the aim of providing a global, unifying view on the data in enterprise information integration, then existing conceptual database schemas should be linked against the ontology, and this connection can fruitfully be exploited for ontology design. Also, typical queries can be evaluated against the ontology already at design time to detect possible modelling problems. More details on these issues can be found in [28].

References

1. G. Antoniou and K. Kehagias. A note on the refinement of ontologies. *International J. of Intelligent Systems*, 15:623–632, 2000.
2. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.
3. F. Baader and R. Küsters. Nonstandard inferences in description logics: The story so far. In *Mathematical Problems from Applied Logic I*, volume 4 of *International Mathematical Series*, pages 1–66. Springer, 2005.
4. F. Baader, B. Sertkaya, and A.-Y. Turhan. Computing the least common subsumer w.r.t. a background terminology. In *Proc. of DL2004*, CEUR-WS, 2004.
5. A. Borgida, E. Franconi, and I. Horrocks. Explaining ALC subsumption. In *ECAI'00*, pages 209–213. IOS Press, 2000.
6. R. Cote, D. Rothwell, J. Palotay, R. Beckett, and L. Brochu. The systematized nomenclature of human and veterinary medicine. Technical report, SNOMED International, Northfield, IL: College of American Pathologists, 1993.

7. B. Cuenca-Grau, I. Horrocks, O. Kutz, and U. Sattler. Will my ontologies fit together? In *Proc. of DL06*, number 189 in CEUR-WS, 2006.
8. J. Gennari, M. Musen, R. Fergerson, W. Grosse, M. Crubezy, H. Eriksson, N. Noy, and S. Tu. The evolution of protege: an environment for knowledge-based systems development. *International J. of Human-Computer Studies*, 58(1):89–123, 2003.
9. S. Ghilardi, C. Lutz, and F. Wolter. Did I damage my ontology? In *Proc. of KR2006*, pages 187–197, 2006.
10. V. Haarslev and R. Möller. RACER system description. In *Proc. of IJCAR 2001*, 2001.
11. I. Horrocks, P. F. Patel-Schneider, and F. van Harmelen. From SHIQ and RDF to OWL: The making of a web ontology language. *J. of Web Semantics*, 1(1):7–26, 2003.
12. A. Kalyanpur, B. Parsia, E. Sirin, and B. Cuenca-Grau. Repairing unsatisfiable concepts in owl ontologies. In *Proc. of ESWC06*. Springer, 2006.
13. K. Lee, T. Meyer, J. Pan, and R. Booth. Computing maximally satisfiable terminologies for the description logic $\mathcal{ALC}$ with GCIs. In *Proc. of DL06*, CEUR-WS, 2006.
14. C. Lutz, D. Walther, and F. Wolter. Conservative extensions in expressive description logics. submitted.
15. T. Maibaum. Conservative extensions, interpretations between theories and all that! In *Proc. of 7th CAAP/FASE Conf. on Theory and Practice of Software Development*, pages 40–66, 1997.
16. D. L. McGuinness. *Explaining Reasoning in Description Logics*. PhD thesis, Department of Computer Science, Rutgers University, Oct. 1996.
17. D. L. McGuinness and A. Borgida. Explaining subsumption in description logics. In *Proc. of IJCAI'95*, pages 816–821, 1995.
18. T. Meyer, K. Lee, and R. Booth. Knowledge integration for DLs. In *Proc. of AAAI'05*, pages 645–650, 2005.
19. I. Niles and A. Pease. Towards a standard upper ontology. In *Proc. of FOIS*, pages 2–9, 2001.
20. A. Rector, S. Bechhofer, C. A. Goble, I. Horrocks, W. A. Nowlan, and W. D. Solomon. The GRAIL concept modelling language for medical terminology. *Artificial Intelligence in Medicine*, 9:139–171, 1997.
21. S. Schlobach. Debugging and semantic clarification by pinpointing. In *Proc. of ESWC05*, pages 226–240. Springer, 2005.
22. S. Schlobach and R. Cornet. Non-standard reasoning services for the debugging of description logic terminologies. In *Proc. of IJCAI 2003*, Acapulco, Mexico, 2003. Morgan Kaufmann, Los Altos.
23. E. Sirin and B. Parsia. Pellet system description. In *Proc. of DL'06*, 2006.
24. K. A. Spackman. Managing clinical terminology hierarchies using algorithmic calculation of subsumption: Experience with SNOMED-RT. *J. of the American Medical Informatics Association*, 2000. Fall Symposium Special Issue.
25. D. Tsarkov and I. Horrocks. FaCT++ Description Logic Reasoner: System Description. Proc. of IJCAR-06. To Appear.
26. The Gene Ontology Consortium. Gene Ontology: Tool for the unification of biology. *Nature Genetics*, 25:25–29, 2000.
27. The TONES consortium. Deliverable 01: State of the art survey, 2005.
28. The TONES consortium. Deliverable 05: Tasks for ontology design and maintenance, 2005.